API of common Python Functions

Short Python function/method descriptions:

__builtins__ :

input([prompt]) -> str
Read a string from standard input. The trailing newline is stripped. The prompt string, if given, is printed without a trailing newline before reading.

len(x) -> int
Return the length of the list, tuple, dict, or string x.

open(name[, mode]) -> file open for reading, writing, or appending
Open a file. Legal modes are 'r' (read), 'w' (write), and 'a' (append).

print(value) -> None
Prints the value.

range([start], stop, [step]) -> list-like-object of int
Return the integers starting with start and ending with stop - 1 with step specifying the amount to increment (or decrement).
If start is not specified, the list starts at 0. If step is not specified, the values are incremented by 1.

importing:

from file_name import function_name
Import function with name function_name from file with name file_name

functions:

def function_name(args):

```
    """docstring describing function here"""
    #function body here
```

for loops:

```
  for obj in list:
    #for loop with obj here
```

Loop over each object obj in list (list can be a range of ints)

while loops:

```
  while condition:
    #while loop body here
```

Loop until condition is not met

if/else:

```
  if condition1:
    #do if condition1 is met
  elif condition2
    #do if condition1 is not met, but condition2 is met
  else
    #do if none of the above conditions are met
```

doctest:

```
  import doctest
  def func_to_test(val):
    """docstring describing function
    >>> func_to_test(testval_1)
    expected output
    >>> func_to_test(testval_2)
    expected output
    ...
    """
    #function body here
  doctest.testmod()
```

Tests that func_to_test returns the expected outputs for values testval_1 and testval_2

unit test:

```
  import unittest
  class SomeTestCase(unittest.TestCase):
    def test_some_value(self):
      self.assertTrue(function_to_test(val_to_test))
   unittest.main()
```

Tests function_to_test using value val_to_test, and asserts that this will return True.
Common assertions include:

assertEqual
assertNotEqual
assertTrue
assertFalse

dict:

```
  D[k] --> object
```

Produce the value associated with the key k in D.

```
  del D[k]
```

Remove D[k] from D.

```
  k in d --> bool
```

Produce True if k is a key in D and False otherwise.

```
  D.get(k) -> object
```

Return D[k] if k in D, otherwise return None.

D.keys() -> list-like-object of object
Return the keys of D.
D.values() -> list-like-object of object
Return the values associated with the keys of D.
D.items() -> list-like-object of tuple of (object, object)
Return the (key, value) pairs of D, as 2-tuples.

safely open a file:

with open('filename.txt') as F:
#Do something with F
Ensures exit method close() will be called on F even if any error happens in the block

file open for reading:

for line in F:
#Do something with line
Loop through each line of the file

file open for writing:

F.write(x) -> int
Write the string x to file F and return the number of characters written.

list:

x in L --> bool
Produce True if x is in L and False otherwise.
L.append(x) -> None
Append x to the end of the list L.
L.index(value) -> int
Return the lowest index of value in L.
L.insert(index, x) -> None
Insert x at position index.
L.remove(value) -> None
Remove the first occurrence of value from L.
L.sort() -> None
Sort the list in ascending order *IN PLACE*.
L[start:end] -> list
Slices the list starting at the start index, until (excluding) the end index

str:

x in s --> bool
Produce True if and only if x is in s.
str(x) -> str
Convert an object into its string representation, if possible.
S.find(sub[, i]) -> int
Return the lowest index in S (starting at S[i], if i is given) where the
string sub is found or -1 if sub does not occur in S.
S.index(sub) -> int
Like find but raises an exception if sub does not occur in S.
S.split([sep]) -> list of str
Return a list of the words in S, using string sep as the separator and
any whitespace string if sep is not specified.
S.strip([chars]) -> str
Return a copy of S with leading and trailing whitespace removed.
If chars is given and not None, remove characters in chars instead.

main block:

```
  if __name__ == '__main__':
#main block code here
```

CSC 148

Python Visualiser: http://www.pythontutor.com/visualize.html#mode=edit
Type Contract:
@type parameter: type
@rtype: type (return type)

Docstring: """
– Describe what the method does, very precisely.
– Mention every parameter by name.
– Do not discuss how the method works (e.g Do not discuss local variables, helper methods, or algorithms.)—Uses for a docstring:
Guides you in writing the body & Defines the interface so callers of the function know how to use it.
If arguments to the function must satisfy requirements/preconditions (other than type contract), say so.

A precondition for a function is a boolean expression that describes restrictions on the values of the arguments.
If the arguments satisfy the type contract and the preconditions:
The function will halt without crashing, and do what the docstring says. Otherwise, no guarantees.

Design by contract
Design by contract is a way of thinking about programming. Each function/method provides a service.
Its service is specified as a contract by the docstring, including: What it expects (the precondition) and What it guarantees (what the function does)

Uses for a function's contract:
• Guides you in writing the body.
• Defines an interface for callers of the function.
• Aids in debugging.
• Aids in code maintenance.
– If you improve the implementation, you know what you must ensure is still guaranteed.
• Helps you argue that the code is correct.
– Defines the standard against which to measure.
– Is the basis for a proof of correctness.

```
w1 = 'words'

# w1 is the name of a variable that refers to some data, which we call an object.
```

An object(name of a variable) has a value, an identity and a type.
All data stored in a Python program has three components: id(where the data is stored in memory), type(determines what functions can operate on the data), and value.
All programming languages have the concept of variables, which can *refer* to data (but are not actually the data themselves).

```
# We can see the value:
>>> w1
'words'
# The identity is a unique value (i.e. no other object has the same identity).
# Often Python uses the memory address of the object as its id, but it doesn't have to. It just has to guarantee uniqueness.
# We can see the id:
>>> id(w1)
4297547872
# We can see the type:
>>> type(w1)
<class 'str'>

# It is the object that has a type.
# w1 itself has no type.  This is very different from many other languages.
# In fact, Python doesn't mind if we have it refer now to a different type of object, although we should have a really good reason for doing this.
>>> w1 = 42
```

```
>>> type(w1)
<class 'int'>
```

When we want to do something with an object, we usually call a method on it.
>>> w1.index('s')
We can get help on a type, which will show us the available methods.
>>> help(str)

Mutability:

Some types of objects(data types) are **immutable** (int, string, boolean) — value stored in the data cannot change.
Once you have a str, you cannot change it. But you can always make a new str and reference that.

```
>>> prof = 'Diane'
>>> id(prof)
4405312456
>>> prof = prof + ' Horton'
>>> prof
'Diane Horton'
# The old str object couldn't change, so Python made a new str object for
# the variable prof to refer to.
>>> id(prof)
4405308016
```

Some types of objects are **mutable** (list, dictionary, and user-defined class).

```
>>> ages = [18, 35, 9, 36, 68]
>>> id(ages)
4405224904
>>> ages.append(1)
>>> ages
[18, 35, 9, 36, 68, 1]
# The list object COULD change. The variable ages is still referencing the same object, but the object has
changed.
>>> id(ages)
4405224904
```

Two Types of Equality:

== compares the values stored at variables; *is* compare the IDs of the respective data

```
# For w3, we use "slicing" to produce a value for it: every character in w1 from index 1 until the end
(that's what happens when we slice with no upper limit). When we slice, we are creating a new string
object.
w1 = 'words'
w2 = 'swords'
w3 = w2[1:]
# Let's check the 3 values.
w1
'words'
w2
'swords'
w3
'words'
# Of course, w1 and w2 are not the same.
w1 == w2
False
# But we might say that "w1 and w3 are the same."
# Python gives us two ways to ask whether they are the same.
w1 == w3
True
w1 is w3
False
# Double equals checks whether they have the same value, but "is" checks whether they are the same object.
# It was sloppy to say above that "w1 and w3 are the same".
# More precise would be to say that "w1 and w3 each reference a string object with the value 'words'" or
"w1 references a string object with the value 'words' and so does w3, but they are two different string
objects."
```

When we create two different objects that have the same value, and it is of an immutable type (int, string, boolean), Python will save some memory and just create one object (2 objects with the same id).

Aliasing

When two variables refer to the same object/data, they are aliases of each other.

Aliasing allows "action at a distance": modification of a variable's value without explicit mention of that variable.

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
# x and z both reference the same object: a list object with [1, 2, 3] as its value. As a result, they
have the same id.
>>> id(x)
4401298824
>>> id(z)
4401298824
# We say that x and z are "aliases".
# Make sure that you understand what "z = x" does, and how it leads to aliasing.

# In contrast, x and y are not aliases.  They each refer to a list object
# with [1, 2, 3] as its value, but they are two different list objects.
# This is reflected in their different ids.
>>> id(x)
4401298824
>>> id(y)
4404546056

>>> x is y
False
>>> x is z
True
>>> x == y
True
>>> x == z
True
```

Aliasing + Mutation

```
# Example 1: y unchanged, id(y) unchanged, but id(x) changed
>>> x = 3
>>> y = 3
>>> x = 5

# Example 2: y unchanged, id(x) & id(y) unchanged and unequal
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> x[1] = 100

# Example 3: y unchanged and equals to old x, id(y) unchanged and equals to old id(x), id(x) changed
>>> x = 3
>>> y = x
>>> x = 5
# After the first two lines, both x and y refer to the same data (with value 3). The third line does not
change the value stored for that data (ints are immutable) but rather changes where x refers to. You can
always change a reference in Python, but only some of the time can change the value stored at a particular
place, depending on the data type.

# Example 4: y changed, id(x) & id(y) unchanged and equal
>>> x = [1, 2, 3]
>>> y = x           # make y refer to the data that x refer to
>>> x[1] = 100      # mutate the value of x, so does value of y (two variables refer to the same data)
```

Function Design Recipe (function docstring)

Let others know how to use the function, describes what the function does (through text and examples) and requirements necessary to use the function.

One such requirement is the **type contract**: this requires that when someone calls the function, they should do so with arguments of a specified type.

Precondition: any property that the function's arguments must satisfy to ensure that the function works as intended.

CSC148: Intro to CS

Encapsulation
Composition, inheritance, and delegation
Polymorphism

Ramp Session

programs are stored in .py files
From the command line: python myfile.py
edit and run program using an IDE (integrated Dev Environment)
Set Up Environment: mark directory as “source root”

1.Using Python Console
import all the functions from a file >>> from file_name import function_name
press UP key to view past commands
Python can also be run interactively, from within PyCharm, or by python or python 3 on command line.
The result is automatically shown (unlike in a program, where you must call print)

2.Python file (.py):
import names from other modules + define classes and functions + main block

3.Python Features:
Interpreted(no compilation necessary)
Object-oriented(classes, inheritance)
Whitespace matters(4 spaces/1 tab for indentation)
No end-of-line character(no semicolons!)
No extra code needed to start(no "public static...")
Dynamically typed(a function can take multiple different types, have different behaviours)
type contracts provide type safety
python wont prevent people from using the wrong types, but at least you warned them!
Strongly typed(all values have a type)
Comments start with a '#' character.

4.**Documentation & Sources**:
Official Python Documentation: http://docs.python.org/py3k/library/
Python Standard Library: https://docs.python.org/3/library/index.html
in python console:
>>> help(print) provides usage information
>>> dir(str) shows names within a given type, module, object

Software Carpentry: http://software-carpentry.org/
Google: http://lmgtfy.com/?q=python+add+to+list

5.**Style Guide**:
Python's style guide: http://www.python.org/dev/peps/pep-0008/
Google's Python style guide: http://google-styleguide.googlecode.com/svn/trunk/pyguide.html
Expert mode: pylint: https://www.pylint.org

6.**Variables** (storing data) Variables refer to an object of some type
①Boolean: True False Operator: and or not
②Integer (whole number): int
Floating-point (decimal) number: float Operator: + - * / %(Modulus) **(Exponent) // (Floor Division)
③None: NoneType (Python’s Null)
④String: immutable lists of characters str
—slices return substrings

```
>>> welcome = 'Hello, world'
>>> welcome[1]      # index into string, slice with [start: end]
'e'
>>> welcome[:3]     # start defaults to 0
'Hel'
>>> welcome[3:]     # end defaults to None
'lo, world'
>>> welcome[:-1]    # index/slice with negative
'Hello, worl'
>>> welcome[::-1]   # reverse a string
'dlrow ,olleH'
```

—concatenation

```
>>> a = 'a'
>>> b = 'b'
>>> a+b
'ab'
```

—len function:

```
>>> len('hello')
5
```

—(can reassign variables to a new string value) ???

—String Formatting: str.format

{} are replaced by the arguments to format

```
>>> n = 1
>>> where = 'here'
>>> '{} bottles {}'.format(n, where)                % {} is a placeholder
'1 bottles here'
```

—Empty strings exist, not the same as None: Empty_string=''

⑤**Type Conversion**: how to sanitize user input int(), float(), str(), bool()

str(5.33) = '5.33'

7.**Sequence**: list & tuple, things in an order

①**List**: a mutable sequence of any objects index and slice like strings

Method: L.append(value) L.remove(value) print(L) L + [1, 2] (concatenation)

```
>>> L = [42, 3.14, 'carpe diem']
>>> L[0] = 'replaced'
>>> L
['replaced', 3.14, 'carpe diem']
>>> L[1]            # index returns the element
3.14
>>> L[2:]           # slice returns a sublist
['carpe diem']
>>> len(L)          # size
3
>>> 3.14 in L       # membership testing
True
>>> L.pop(2)        # remove/return val at [index]
'carpe diem'
>>> L
['replaced', 3.14]
>>> L + [1, 4]             # list concatenation
['replaced', 3.14, 1, 4]
```

Careful! MULTIPLE variables might be referring to the same mutable data structure.

②**Tuple**: fast, simple lists, immutable

```
>>> tuple = (1, 3.14, 'car')
>>> tuple[0]
1
>>> tuple[0] = 'a'
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> L = list(tuple)              # can always create a list from tuple
```

8.**Dictionary**: dict are an unordered association of keys with values {'key':value}, keys must be unique and immutable

```
>>> score = {'A': 90, 'B': 75, 'C': 60}
>>> score['A']              # get
90
>>> score['D'] = 50         # set
>>> score.pop('B')          # delete
75
>>> 'C' in score            # membership testing
True
>>> for key in score:             # loop over keys
...     print("{}: {}".format(key, score[key]))
...
D: 50
C: 60
A: 90
```

9.**For Loop**: repeats some code for each element in a sequence

for key in score

—Use **range**(n) or range(a, b) in a for loop to loop over a range

```
>>> for i in range(2):
...     print(i)
...
0
1
>>> for i in range(4,6):
...     print(i)
...
4
5
>>> color = ['red', 'blue', 'black']
>>> for i in range(len(color)):
...     print('{} and {}'.format(i, color[i]))
...
0 and red
1 and blue
2 and black
```

For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use. The only difference is that range returns a Python list object and xrange returns an xrange object. It means that xrange doesn't actually generate a static list at run-time like range does. It creates the values as you need them with a special technique called yielding. This technique is used with a type of object known as generators.

—**zip** returns a list of pairs

```
>>> for (i, color) in zip(range(len(color)), color):
...     print('{} and {}'.format(i, color))
...
0 and red
1 and blue
2 and black
```

```
    pairs = zip(['a', 'b'], [1, 2, 3])
    for i, color in zip(['a', 'b'], [1, 2, 3]):
        print('{} and {}'.format(i, color))    # a and 1 \ b and 2

    print(pairs)          # <zip object at 0x102183688>   zip returns an object

    tuples = [('a', 1), ('b', 2), ('c', 3)]
    x, y = zip(*tuples)
    print(x, y)     # ('a', 'b', 'c') (1, 2, 3)
```

—**enumerate** assigns a number to each element

```
>>> for (i, color) in enumerate(color):
...     print('{} and {}'.format(i, color))
...
0 and red
1 and blue
2 and black
```

10.**While Loop**: keep repeating a block of code while a condition is True
—break can be used to exit a loop early
—continue and pass work too

11.**Conditionals** (if, elif, else):
—if statement allows you to execute code sometimes (based upon some condition)
—elif(else if) and else are optional

12.**Functions**:
—allows to group together a bunch of statements into a block that you can call
—take in information(arguments) and give back information(return value)
—If no return is specified, None will be returned.
—Functions can modify mutable arguments

Function Design Recipe

①Example Calls: doctest
Docstring: the part of function in triple-quotes, which is shown when help is called on function
~a multi-line, meaning triple-quoted, string right after the declaration,
~describe what the function does, not how it does it
~describe the argument and return types
~shown when help is called on the function, it should be sufficient for other people to know how to use this function
②Type Contract: argument(s) and return values
@type args: int/float
@rtype: None (if no return value)
③Header:
def function(args):
④Descriptions: what the function does, not how it does it
⑤Body: actual function code
⑥Test your function
Function Control Tools:
break (break out of the loop)
pass (a null operation, nothing happens)
continue (skip to next iteration of loop)

Functions can modify mutable(list..) arguments.

13.**Modules**: Python has a spectacular assortment of modules that you can use (first import their names)

13.Memory & Mutability:
If you assign a variable to an immutable object (like a string), and then change where the variable refers to, a whole new object is created in memory, and the variable points to that new object.
If you assign a variable to a mutable object (like a list), and then change the value that the variable refers to, the variable will continue to point to the the original object, which has been internally changed.
http://www.pythontutor.com/ visualize.html
Variable Aliasing—Mutable

```
>>> sorted_list = [1, 2, 3]
>>> not_a_copy = sorted_list  # not a copy
>>> not_a_copy.append(0)
>>> sorted_list
```

```
[1, 2, 3, 0]  # crap
>>> actually_a_copy = list(sorted_list)
>>> another_copy = sorted_list[:]

>>> sorted_list = [1, 2, 3]
>>> not_a_copy = sorted_list  # not a copy
>>> not_a_copy = [4, 5, 6]
>>> sorted_list
[1, 2, 3]  # nope we're still OK!
•We assigned not_a_copy to an entirely new list, which changed which object it was referring to
•sorted_list will still refer to the original list

def double(L):
       """ Modify L so that it is equivalent to L + L
       @type L: list
       @rtype: list
       """
       for i in range(len(L)):
              L.append(L[i])

L = [1, 2, 3]
L = double(L) # Don't do this! Why? # double(L) changes the list and then returns None
print(L) # We expect[1, 2, 3, 1, 2, 3]

def double(L):
      """ Create a new list that is equivalent to L + L
      @type L: list
      @rtype: list
      """
     newList = []
     for j in range(2):
         for i in range(len(L)):
             newList.append(L[i])
     return newList
L = [1, 2, 3]
endList = double(L)
print (endList) # [1, 2, 3, 1, 2, 3]
print(L) # [1, 2, 3]
```

More about this:

```
a = [0, 1, 2, 3, 4]
b = a               # assign b to the same address as a, use a.copy() to copy
b[2] = 10
c = a[1]            # integers 就是copy
c = 20
d = [5, 6, 7, 8]
d = b
```

14.**Debugger**: Run-Debug
first add breadline

15.**Standard Input/Output**:

①Generating output(stdout): print()
—can take multiple arguments (will be joined with spaces)
—does not return a value
②Reading keyboard input: input()
③Open files:
use with/as to open something for a while, but always close it, even if something goes wrong

```
with open('name.txt') as file_name:
       for line in file_name:
              Do sth. with each line
```

④Write files:

```
with open('output.txt', 'w') as file_name:
```

```
        file_name.write('haha \n')
```

16.Test the Code:
–Assures correctness of the program under specific conditions
–Thinking of testing while coding makes the coder write code that is better designed
–Helps you think about edge cases (e.g. What if user tries to delete a file that isn't there? What if a function that takes mutable data is given an immutable type?)

①Doctest: Informs others on how to expect your function to be used / the edge cases they may encounter

```
if __name__ == '__main__':
        import doctest
        doctest.testmod()
```

②Unittest: Able to run tests in a separate file which allows you to run more without worrying about cluttering the docstring

```
import unittest
from even import is_even

class EvenTestCase(unittest.TestCase):
        def test_is_two_even(self):
                self.assertTrue(is_even(2))
class OddTestCase(unittest.TestCase):
        def test_is_three_odd(self):
                self.assertFalse(is_even(3))

if __name__ == '__main__':
        unittest.main()
```

Week 1 · Function Design Recipe & Class Design Recipe

—understand and write a solution for a real-world problem (get specification in text, turn them into a solution, in step-by-step fashion)
—abstract data type (ADTs) to represent and manipulate info (reveal the API, but hide implementation details)
—recursion: functions that call themselves
—exception: detect unusual situations
—design: how to structure a program (well-documented, easily-modified code)
—efficiency: how much resource (time/space) does a program use (some code is technically correct, but not feasible)

NOTE: Credit to/Citation if use other ideas

Function Design Recipe

Method: a function defined in a class

①**Example:** Write some examples of calls to your function1 and the expected returned values. Include an example of a standard case (as opposed to a tricky or corner case.) Put the examples inside an indented triple-quoted string.
②**Type Contract:** Write a type contract that identifies name and type of each parameter. Choose a meaningful name for each parameter. Also identify the return type of the function. Put the type contract above the examples.
③**Header:** Write the function header above the docstring and outdent it.
④**Description:** In the same line as the opening triple-quote mark, put a one-line summary of what the function does. If necessary, you can put an optional, longer description above the type contract. Mention each parameter by name.
· **Precondition:** a Boolean expression that describes restrictions on the values of the arguments.
· **Representation Invariant:**
a) A Boolean statement about the attributes of the class
b) Which must be True for any instance of the class
c) The type contract is a simple representation invariant

⑤**Body:** Write the body of the function by remembering to indent it to match the docstring. To help yourself write the body, review your example cases from step 1 and how you determined the return values. You may find it helpful to write a few more example calls in the docstring.

```
def length_is_multiple(string, num):
        """Return whether num evenly divides length of string.
        @param str string: a string
        @param int num: a whole number
        @param num: int
                    a whole number
        @rtype: bool
        >>> length_is_multiple("two", 2)
        False
        >>> length_is_multiple("two", 3)
        True
        """
        return len(string) % num == 0
```

```
# Header
# Description
# Type Contract


# Example, doc test




# Body
```

⑥**Test Your Function:**
Test your function on all your example cases including any additional cases you created in step 5. Additionally try it on extra tricky or corner cases. In order to automatically test your docstring examples you can include the following at the end of the file:

```
   if __name__ == '__main__':
       import doctest
       doctest.testmod()
```

Class/Function Design Recipe: docstring requirements
if return type is not None, method is written by us, example is required.

Build class

Define a class API (Application Program Interface):
• choose a class name (capitalized first letter) and write a brief description in the class docstring
• write some examples of client code that uses your class
• decide what services your class should provide as public methods, for each method declare an API (examples, header, type contract, description)
• decide which attributes your class should provide without calling a method, list them in class docstring

Implement the class:
• body of special methods: __init__, __eq__, __str__ (double underscore)
  constructor (__init__): initialize every instance attribute
• body of other methods:
• testing

```
p = Point(attributes)             # p is an object/instance of type Point.

Point.__dict__
p.__dict__


>>> type(3)
<class 'int'>

def __str__(self):
       """ Return a string representation of Point self."""
       return "{} and {}".format(self.x, self.y)

def __eq__(self, other):
       """ Return whether self is equivalent to other."""
       return ((type(self) == type(other)) and (self.x == other.x) and (self.y == other.y))
```

Element of a class: instance attributes (data), methods (operations), representation invariants (properties).

Information Hiding: designing class by separating the public interface of the class from the implementation details.

Week 2 · Docstring, Composition, Private Method

build-in integers, floats etc.

1.

2.write some examples of client code that uses your class

3.decide what services your class should
special methods: __XX__

```
__eq__, __init__,
__str__ (print something)
```

Fix method num_items, rather than the usual way of calling a method q.num_items(), we want to be able to call it as len(q). Ans: Change the method's name to __len__

```
4.decide which attributes your class should provide without calling a method, list them in the class
docstring.
num, den
```

Private Method/Data: begins with a leading underscore _, use such a method/data when we do not want programs external to our class to call it. (helper methods in a class)

docstring are public interface; private method are not be used outside class, so no docstring
Private Method: use # inside of """ """

```
docstring:
>>> Command
returned value
```

Inheritance vs **Composition**

```
def Subclass(SuperName):                        # superclass name is SuperName
      def __init__(self, arguments):            # constructor
            SuperName.__init__(self)            # or super().__init__(self)
            self.argument = argument            # extended
            self.method2 = method2              # @type method2: Callable[[Object], argument]

      def method(self, arguments):
            SuperName.method(self, arguments)

      def method2(self, argument):
            …
            return …
```

Note: two ways to call a method from a parent class: SuperName.__init__(self, argument) super().__init__(self)

Most of the uses of inheritance can be simplified or replaced with composition.

Inheritance

Inheritance is used to indicate that one class will get most or all of its features from a parent class. This happens implicitly whenever you write `class Child(Parent)`, which says "Make a class `Child` that inherits from `Parent`." When you do this, the language makes any action that you do on instances of `Child` also work as if they were done to an instance of `Parent`. Doing this lets you put common functionality in the `Parent` class, then specialize that functionality in the `Child` class as needed.

A child class inherits all methods of its parent. A child class can extend the parent by adding attributes or methods. A child class can override the parent, by re-defining an existing method.

When you are doing this kind of specialization, there are three ways that the parent and child classes can interact:

1. Actions on the child imply an action on the parent.
2. Actions on the child override the action on the parent.
3. Actions on the child alter the action on the parent.

①Implicit Inheritance

First I will show you the implicit actions that happen when you define a function in the parent, but *not* in the child.

```
class Parent(object):

    def implicit(self):
        print "PARENT implicit()"

class Child(Parent):
    pass
```

```
dad = Parent()
son = Child()

dad.implicit()
son.implicit()
```

The use of `pass` under the `class Child`: is how you tell Python that you want an empty block. This creates a class named `Child` but says that there's nothing new to define in it. Instead it will inherit all of its behavior from `Parent`. When you run this code you get the following:

```
$ python ex44a.py
PARENT implicit()
PARENT implicit()
```

Notice how even though I'm calling `son.implicit()` on line 16, and even though `Child` does *not* have a `implicit` function defined, it still works and it calls the one defined in `Parent`. This shows you that, if you put functions in a base class (i.e., `Parent`) then all subclasses (i.e., `Child`) will automatically get those features. Very handy for repetitive code you need in many classes.

②Override Explicitly

The problem with having functions called implicitly is sometimes you want the child to behave differently. In this case you want to override the function in the child, effectively replacing the functionality. To do this just define a function with the same name in `Child`.

```
class Parent(object):

    def override(self):
        print "PARENT override()"

class Child(Parent):

    def override(self):
        print "CHILD override()"

dad = Parent()
son = Child()

dad.override()
son.override()
```

In this example I have a function named `override` in both classes, so let's see what happens when you run it.

```
$ python ex44b.py
PARENT override()
CHILD override()
```

As you can see, when line 14 runs, it runs the `Parent.override` function because that variable (`dad`) is a `Parent`. But when line 15 runs it prints out the `Child.override` messages because `son` is an instance of `Child` and `Child` overrides that function by defining its own version.

③Alter Before or After

The third way to use inheritance is a special case of overriding where you want to alter the behavior before or after the `Parent` class's version runs. You first override the function just like in the last example, but then you use a Python built-in function named `super` to get the `Parent` version to call.

```
class Parent(object):

    def altered(self):
        print "PARENT altered()"

class Child(Parent):

    def altered(self):
        print "CHILD, BEFORE PARENT altered()"
        super(Child, self).altered()
        print "CHILD, AFTER PARENT altered()"

dad = Parent()
son = Child()
```

```
dad.altered()
son.altered()
```

The important lines here are 9-11, where in the `Child` I do the following when `son.altered()` is called:

—Because I've overridden `Parent.altered` the `Child.altered` version runs, and line 9 executes like you'd expect.

—In this case I want to do a before and after so after line 9, I want to use `super` to get the `Parent.altered` version.

—On line 10 I call `super(Child, self).altered()`, which is aware of inheritance and will get the `Parent` class for you. You should be able to read this as "call `super` with arguments `Child` and `self`, then call the function `altered` on whatever it returns."

—At this point, the `Parent.altered` version of the function runs, and that prints out the `Parent` message.

—Finally, this returns from the `Parent.altered` and the `Child.altered` function continues to print out the after message.

If you run this, you should see this:

```
$ python ex44c.py
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

④All Three Combined:

To demonstrate all of these, I have a final version that shows each kind of interaction from inheritance in one file:

```
class Parent(object):

    def override(self):
        print "PARENT override()"

    def implicit(self):
        print "PARENT implicit()"

    def altered(self):
        print "PARENT altered()"

class Child(Parent):

    def override(self):
        print "CHILD override()"

    def altered(self):
        print "CHILD, BEFORE PARENT altered()"
        super(Child, self).altered()
        print "CHILD, AFTER PARENT altered()"

dad = Parent()
son = Child()

dad.implicit()
son.implicit()

dad.override()
son.override()

dad.altered()
son.altered()
```

Go through each line of this code, and write a comment explaining what that line does and whether it's an override or not. Then run it and confirm you get what you expected:

```
$ python ex44d.py
PARENT implicit()
PARENT implicit()
PARENT override()
CHILD override()
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

The Reason for `super()`

This should seem like common sense, but then we get into trouble with a thing called multiple inheritance. Multiple inheritance is when you define a class that inherits from one or *more* classes, like this:

```
class SuperFun(Child, BadStuff):
    pass
```

This is like saying, "Make a class named `SuperFun` that inherits from the classes `Child` and `BadStuff` at the same time."

In this case, whenever you have implicit actions on any `SuperFun` instance, Python has to look-up the possible function in the class hierarchy for both `Child` and `BadStuff`, but it needs to do this in a consistent order. To do this Python uses "method resolution order" (MRO) and an algorithm called C3 to get it straight.

Because the MRO is complex and a well-defined algorithm is used, Python can't leave it to you to get the MRO right. Instead, Python gives you the `super()` function, which handles all of this for you in the places that you need the altering type of actions as I did in `Child.altered`. With `super()` you don't have to worry about getting this right, and Python will find the right function for you.

Using super() with __init__

The most common use of `super()` is actually in `__init__` functions in base classes. This is usually the only place where you need to do some things in a child, then complete the initialization in the parent. Here's a quick example of doing that in the `Child`:

```
class Child(Parent):

    def __init__(self, stuff):
        self.stuff = stuff
        super(Child, self).__init__()
```

This is pretty much the same as the `Child.altered` example above, except I'm setting some variables in the `__init__` before having the `Parent` initialize with its `Parent.__init__`.

Composition

Inheritance is useful, but another way to do the exact same thing is just to *use* other classes and modules, rather than rely on implicit inheritance. If you look at the three ways to exploit inheritance, two of the three involve writing new code to replace or alter functionality. This can easily be replicated by just calling functions in a module.

Composition: two classes, one uses instances of the other, one has an attribute which is an instance of the other.

```
class Other(object):
    def override(self):
        print "OTHER override()"

    def implicit(self):
        print "OTHER implicit()"

    def altered(self):
        print "OTHER altered()"

class Child(object):

    def __init__(self):
        self.other = Other()

    def implicit(self):
        self.other.implicit()

    def override(self):
        print "CHILD override()"

    def altered(self):
        print "CHILD, BEFORE OTHER altered()"
        self.other.altered()
        print "CHILD, AFTER OTHER altered()"

son = Child()

son.implicit()
son.override()
```

```
son.altered()
```

In this code I'm not using the name `Parent`, since there is *not* a parent-child `is-a` relationship. This is a `has-a` relationship, where `Child has-a Other` that it uses to get its work done. When I run this I get the following output:

```
$ python ex44e.py
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
OTHER altered()
CHILD, AFTER OTHER altered()
```

You can see that most of the code in `Child` and `Other` is the same to accomplish the same thing. The only difference is that I had to define a `Child.implicit` function to do that one action. I could then ask myself if I need this `Other` to be a class, and could I just make it into a module named `other.py`?

When to Use Inheritance or Composition

The question of "inheritance versus composition" comes down to an attempt to solve the problem of reusable code. You don't want to have duplicated code all over your software, since that's not clean and efficient. **Inheritance solves this problem by creating a mechanism for you to have implied features in base classes. Composition solves this by giving you modules and the ability to call functions in other classes**.

If both solutions solve the problem of reuse, then which one is appropriate in which situations? The answer is incredibly subjective, but I'll give you my three guidelines for when to do which:

—Avoid multiple inheritance at all costs, as it's too complex to be reliable. If you're stuck with it, then be prepared to know the class hierarchy and spend time finding where everything is coming from.

—Use composition to package code into modules that are used in many different unrelated places and situations.

—Use inheritance only when there are clearly related reusable pieces of code that fit under a single common concept or if you have to because of something you're using.

Do not be a slave to these rules. The thing to remember about object-oriented programming is that it is entirely a social convention programmers have created to package and share code. Because it's a social convention, but one that's codified in Python, you may be forced to avoid these rules because of the people you work with. In that case, find out how they use things and then just adapt to the situation.

Week 3 · Inheritance, Unittest & Doctest

Documentation
Inherited Method, Attributes: no need to document again
Extended Method: use the superclass method, then add something
Overridden Method: new docstring, what it does now in subclass

Two ways to call a method from a parent class SuperName.__init__(self) super().__init__(self)

Queue: First-In-First-Out
Stack: Last-In-First-Out

>>> help(ClassName)

Use @param instead of @type ???????

type().__name__. # get the name of class

List Comprehension: create a new list from a list [expression **for** name **in** iterable **if** condition]
>>> old_list
>>> new_list = [x**2 for x in old_list if x>=10]
A comprehension can span several lines, if that makes it easier to understand
Python expressions evaluate to values, name refers to each element of iterable (list, tuple, dictionary,…) in turn, and a Boolean condition evaluates to either True or False.

python List: sequences of items, can be added, removed, accessed by position
python Dictionary: collection of items accessed by their associated keys
Stack: specialized list where we only have access to most recently added item

Operations that stack should support:

class Stack:
    attribute:
    method: add, remove, is_empty

class Sack:

use dictionary:
use strings as key / use tuples as key
/ use integers as key(not good, they will be stored in order)

Abstract Class: class Container (a class with no attributes and no method bodies)
any class with at least one non-implemented method.
—Client code should never construct an **instance** of an abstract class.
—For client code, we know what all instance of any descendant of the abstract class have in common. This allows client code to take an instance an operate on it regardless of which specific kind it is.
    For a child class, the abstract parent defines exactly what the child is responsible for implementing.
—Purpose of writing methods whose body is a single line of code raise NotImplementedError: used to document public interface defined by an abstract class

Test a Function: write calls in the shell, read the results and judge its correctness
Doctest: (Python Library doctest)

```
if __name__ == '__main__':
    import doctest          # import doctest library
    doctest.testmod()       # run the tests
```

Put in a few basic doctests inside docstrings (for readers), and use unittest library to help test your code.

Unittest: unittest library allows to write tests in a separate file, and run them separately.
Each method starting with test is a separate test. They are all run independently of each other, and in a random order.
Tests should use assert* method as the actual actions that verify the correctness of your code.
A complete set of tests are too long to go into the docstring of a function/method, which are meant to be read by users.
    Doctests don't allow us to run tests that read in data from files.
    Doctests don't allow us to use hypothesis property-based tests.

Setup: every time you do the test, you get a fresh new object
TearDown:

set: careless about order, can only have each element once.

Good Unittest: Simple Cases + Typical Cases + Edge Cases

```
import unittest
import file_name

class TestMethodName(unittest.TestCase):
    def test_base_case(self):
        actual = file_name.method_name(input)
        expected = expected_value
        self.assertEqual(actual, expected)
    def test_single_case(self):
    def test_long_single_case(self):
    def test_nested_case(self):


if __name__ == '__main__':
    unittest.main(exit=False)
```

Choose Test Case:

Structure the test around properties of inputs (value-based or structural properties based on the type of input)
—integer: 0, 1, positive, negative, “small”, “large”
—list: empty, length 1, no duplicates, duplicates, sorted, unsorted
—string: empty, length 1, alphanumeric characters only, special characters
For functions that take in multiple inputs, properties defining the relationships between the inputs often play a role.
(For an input of two numbers, we might have a test for when the first is larger than the second, and another for the the second is larger than the first. For an input of one object and a list, we might have a test for when the object is in the list, and another for when the object isn’t)
And finally, keep in mind that these are rules of thumb only; none of these properties will always be relevant to a given function. For a complete set of tests, you must understand *exactly* what the function does, to be able to identify what properties of the inputs really matter.

Week 4 · Recursion

The key idea of recursion is this: identify how an object or problem can be broken down into smaller instances with the same structure. In programming, we exploit the recursive structure of objects and problems by making use of **recursive functions**, which are functions that call themselves in their body.
A nested list is one of two things: a single integer, or a list of other nested lists.

Loop vs Recursion:

```
L1 = [1, 9, 8, 15]          # flat list, list of depth 1
print(sum(L1))

# list with sub lists,
# turn the nested list into two flat list

L2 = [[3, 5, 4], L1] # list of depth 2
sum([(sum(row) for row in L2)])

L3 = [L1, L2, 7, L1]

def sum_list(L):
    """
    Return the sum of all ints in L.

    @param int|list[int|list[...]] L:
            possibly-nested list of ints, finite depth
    >>> sum_list([1, [2, 3], [4, 5, [6, 7], 8]])
    36
    >>> sum([])
    0
    """
    if isinstance(L, list):       # sum() operates on a list of integers
        return sum([sum_list(x) for x in L])    # Recursive Step with list comprehension
    else:
        return L           # Base Case
```

1. What helper methods(function) does this function call?
   isinstance and sum, and itself (sum_list)
   recursion involves calling itself

2. So far, we haven't confirmed that the function works in any cases.
Trace this call: sum_list(27) # trace this call with simplest input
sum_list(27) —> 27

3.Generalization:
Trace of this call: sum_list([4, 1, 8]) a list of depth one
sum_list([4, 1, 8]) —> sum([sum_list(4), sum_list(1), sum_list(8)]) —> sum([4, 1, 8]) —> 13

Trace this call: sum_list([4])
sum_list([4]) —> sum ([sum_list(4)]) —> sum([4]) —> 4

Trace this call: sum_list([])
??? —> 0

Trace this call: sum_list([4, [1, 2, 3], 8]) list of depth two “keep unwrapping”
sum_list([4, [1, 2, 3], 8]) —> sum([sum_list(4), sum_list([1, 2, 3]), sum_list(8)]) —>sum([4, 6, 8]) —> 18
Trace this call: sum_list([[1, 2, 3], [4, 5], 8])
sum_list([[1, 2, 3], [4, 5], 8]) —> sum([sum_list([1, 2, 3]), sum_list([4, 5]), sum_list(8)]) —> sum([6, 9, 8]) —>23
Trace this call: sum_list([1, [2, 2], [2, [3, 3, 3], 2]]) list of depth three
sum_list([1, [2, 2], [2, [3, 3, 3], 2]])
—> sum([sum_list(1), sum_list([2, 2]), sum_list([2, [3, 3, 3], 2])])
—> sum([1, 4, 13]) -> 18
sum([sum_list(2), sum_list([3, 3, 3]), sum_list(2)])

Problem: max([]) does not defined Need another base case (base case that does not use recursion)

The **depth** of a nested list is the maximum number of times a list is nested inside other lists, with the depth of a single integer being 0.
This is a **Recursive/self-referential Definition**: it defined nested lists in terms of other nested lists.
depth([])
1

```
def depth(list):
        if obj == []:           # empty list
                return 1
        elif isinstance(obj, list):
                return 1 + max([depth(x) for x in obj])
        else:           # it is a non-list
                return 0
```

Find the maximum number:
max() is not defined on empty list

functional programming:
<expression 1> if <condition> else <expression 2>
“victory” if a>10 else “defeat” # return directly

```
L = [3, 1, 3]
all([x == 3 for x in L])
False
L = []
all([x == 3 for x in L])
True            # no counter-example

L = [3, 1, 3]
any([x == 3 for x in L])
True
L = []
any([x == 3 for x in L])
False
```

Design "Recipe" for recursive functions
①Identify the recursive structure of the problem, which can usually be reduced to finding the recursive structure of the input. Figure out if it's a nested list, or some other data type which can be expressed recursively.
Once you do this, you can often write down a code template to guide the structure of your code. For example, the code template for nested lists is:

```
def f(obj):
        if isinstance(obj, int):
                ...
```

```
    else:
        for lst_i in obj:
        ... f(lst_i) ...
```

②Identify and implement code for the base case(s). Note that you can usually tell exactly what the base cases are based on the structure of the input. For nested lists, the common base case is when the input is an integer - and if you follow the above template, you won't forget it.

③Write down a concrete example of the function call on an input of some complexity. Then write down the relevant recursive function calls (determined by the structure of the input), and what they output based on the docstring of the function. This is a good place for a table.

④Take your results from step 3, and figure out how to combine them to produce the correct output for the original call. This is usually the hardest step, but once you figure this out, you can implement the recursive step in your code!

a recursive function has a conditional structure that specifies how to combine recursive sub-calls (general case), and when/how to stop (the base case, or cases).

Template for **Structural Recursion**:
recursion when input is a recursive structure:
if input cannot be decomposed into recursive sub-structures, you have a base case and you directly return a result without recursion
if input can be decomposed into recursive sub-structures, solve them recursively and combine the result(s)
this reduces your job to (a)figuring out how to detect whether the input can be decomposed or not, (b)figuring out how what result to return for the base case, and (c)figuring out which substructures to solve recursively and how to combine their solutions

Recursion Template:
if <condition to detect base case>:
    do something without recursion
else: # general case
    do something that involves recursive calls

```
def gather_lists(list_):
    """ Concatenate all the sublists of L and return the result.

    @param list[list[object]] list_: list of lists to concatenate
    @rtype: list[object]

    >>> gather_lists([[1, 2], [3, 4, 5]])
    [1, 2, 3, 4, 5]
    >>> gather_lists([[6, 7], [8], [9, 10, 11]])
    [6, 7, 8, 9, 10, 11]
    """
    new_list = []
    for l in list_:
        new_list += l
    return new_list
```

Week 5 · Exception

Stool 1, 2, 3
n=1 1->3
n=2 1->2, 1->3, 2->3
n=3 1->3, 1->2, 3->2, 1->3, 2->1, 2->3, 1->3

3 Stools Hanoi:

```
def move_3_cheeses(n, source, intermediate, destination):
    if n > 1:
        move_3_cheeses(n - 1, source, destination, intermediate)
```

```
        move_3_cheeses(1, source, intermediate, destination)
        move_3_cheeses(n - 1, intermediate, source, destination)
    else:  # just 1 cheese
        print("{} -> {}".format(source, destination))
```

4 Stools Hanoi:
—> Recursive for A1: 4 Stools, any number of cheeses

Exception:
ValueError
ZeroDivisionError
IndexError

raise

class ExtremeException(Exception):
    pass # inherit build-in

if __name__ ==‘__main__’:
    try:
        1/0

as long as an Exception is raised, all the rest of code does not run

    except SpecialException as se:
        print(se)
        print(‘caught as SpecialException’)

    except Exception as e:
        print(e)
        print(‘caught as Exception’)

    from the most specialized to the least specialized

```
class SpecialException(Exception):
    pass

class ReallySpecialException(SpecialException):
    pass


def exception_raiser(value):
    """ A function to raise exceptions """

    if value < 0:
        raise ReallySpecialException('negative number')
    if value == 0:
        return 10 / value
    if value < 10:
        raise SpecialException('value is less than 10')

def exception_handler(value):
    try:                # try-catch
        exception_raiser(value)      # the code you think might cause an error
        # we won't get to the line below if an exception occurs
        value = 7
    except ReallySpecialException:
        print('that number is negative')
    except SpecialException:
        # what to do if an error occurs
        print('sorry, bad value')
    except ZeroDivisionError:
```

```
        # often this will contain code that does more than print
        value = 1
    except Exception:
        print('something else went wrong')
    else: # code here is executed only if no exception occurs
        print('yay')
    finally: # this is executed no matter what whether or not an exception occurred
        print('no except')

if __name__ == '__main__':
    exception_handler(-1)
```

```
def is_fluffy(word):
    """ Return True iff word is fluffy. """
    for ch in word:
        if ch not in 'fluffy':                # string is used like a list there.
            return False
    return True

def shorter_fluffy(word):
    return all([ch in 'fluffy' for ch in word])
```

Set: unordered collection of elements, no duplicates

```
L = [1, 2, 1, 3, 1, 5, 6, 6]
L = list(set(L))          # to remove duplicates from L
```

set(iterable) iterable: list, dictionary, tuple

Union: s1 | s2 Intersection: s1 & s2 Difference: s1 - s2

Week 6 · Parenthesization, Linked List

Parenthesization: In some situations it is important that opening and closing parentheses, brackets, braces match (balancing).

Balanced Parentheses:
—a string with no parentheses is balanced
—a string that begins with a left parenthesis "(", ends with a right parenthesis ")", and in between has balanced parentheses is balanced. Same for brackets "[...]" and braces "..."
—concatenation of two strings with balanced parentheses is also balanced

Use stack to check whether parentheses of a string is balanced

Linked List

List: allocate large blocks of contiguous memory

Two ways of thinking linked list nodes:
①recursive concept: lists made up of an item(value) and a sub-list(rest):
[12, [99, [3, [2, […]]]]]
②as objects(nodes) with a item(value) and a reference to other similar objects

use next_ instead of next because there is a build-in next in python, do not want to mix up.

5->7->11->13->None

Python has weak support for recursion.

Decide that two LLNodes are __eq__:
entire chains until None are equivalent;
just the values match;
same id.

Optional Parameter: a parameter of a function/method with a default value (immutable value, int/str/None)
in the constructor of LinkedListNode, next_ is an optional parameter.
All optional parameters must appear after all of the required parameters in the function header.

```
""" Node and LinkedList classes """

class LinkedListNode:
    """ Node to be used in linked list

    === Attributes ===
    @param LinkedListNode next_: successor to this LinkedListNode
    @param object value: data this LinkedListNode represents
    """
    def __init__(self, value, next_=None):
        """ Create LinkedListNode self with data value and successor next_.

        @param self: LinkedListNode
              this LinkedListNode
        @param value: object
              data of this linked list node
        @param next_: LinkedListNode|None
              successor to this LinkedListNode.
        @rtype: None
        """
        self.value, self.next_ = value, next_

    def __str__(self):
        """ Return a user-friendly representation of this LinkedListNode.

        @param LinkedListNode self: this LinkedListNode
        @rtype: str

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        s = "{} ->".format(self.value)
        # create a reference to "walk" along the list (current node)
        current = self.next_
        # build string s for each remaining node
        while current is not None:
            s += " {} ->".format(current.value)
            current = current.next_
        return s + "|"

    def __eq__(self, other):
        """ Return whether LinkedListNode self is equivalent to other.

        Assume self and other are not nodes in an looped list.

        @param LinkedListNode self: this LinkedListNode
        @param LinkedListNode|object other: object to compare to self.
        @rtype: bool

        >>> LinkedListNode(5).__eq__(5)
        False
        >>> n1 = LinkedListNode(5, LinkedListNode(7))
        >>> n2 = LinkedListNode(5, LinkedListNode(7, None))
        >>> n1.__eq__(n2)
        True
        """
        # we'll say 2 nodes are equal if the linked lists
        # beginning at those nodes are also equal
        if not type(other) == LinkedListNode:
            return False
        current_node = self
        other_node = other
        while (current_node is not None
               and other_node is not None):     # think of the negation (condition to exit while-loop)
```

```
            if current_node.value != other_node.value:  # sentinel checking
                return False
            # self_node.next_ == other_node.next_ is a recursive call here
            current_node = current_node.next_
            other_node = other_node.next_
        assert current_node is None or other_node is None
        # return True if we reached the end of both lists and False otherwise
        return (current_node is None and other_node is None)
        # exercise: write this recursively (I claim it's much simpler
```

```
class LinkedList:
    """ Collection of LinkedListNodes

    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList
                        a non-negative integer
    """
    def __init__(self):
        """ Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        """
        self.front, self.back, self.size = None, None, 0

    def __str__(self):
        """ Return a human-friendly string representation of LinkedList self.

        @param LinkedList self: this LinkedList

        >>> lnk = LinkedList()
        >>> lnk.prepend(5)
        >>> print(lnk)
        5 ->|
        """
        return str(self.front)

    def __eq__(self, other):
        """ Return whether LinkedList self is equivalent to other.

        @param LinkedList self: this LinkedList
        @param LinkedList|object other: object to compare to self
        @rtype: bool

        >>> LinkedList().__eq__(None)
        False
        >>> lnk = LinkedList()
        >>> lnk.prepend(5)
        >>> lnk2 = LinkedList()
        >>> lnk2.prepend(5)
        >>> lnk.__eq__(lnk2)
        True
        """
        return (type(self) == type(other) and
                self.front == other.front)

    def append(self, value):
        """ Insert a new LinkedListNode with value after self.back.

        @param LinkedList self: this LinkedList.
        @param object value: value of new LinkedListNode
        @rtype: None

        >>> lnk = LinkedList()
        >>> lnk.append(5)
        >>> lnk.size
        1
```

```
        >>> print(lnk.front)
        5 ->|
        >>> lnk.append(6)
        >>> lnk.size
        2
        >>> print(lnk.front)
        5 -> 6 ->|
        """
        # assert ((self.front is None and self.back is None) or
                (self.front is not None and self.back is not None))
        self.size += 1                                          # increment size ☆
        new_node = LinkedListNode(value)                        # create new node ☆
        if self.front is None:  (if self.size == 0)             # if list empty, update front ☆
            assert self.back is None and self.size == 1
            self.front, self.back = new_node, new_node
            # assign the tuple on the right to tuple on the left
        else:               # !!! careful about the order of assignment
            assert self.back is not None
            self.back.next_ = new_node                          # old back refers to new back ☆
            self.back = new_node                                # change what back refers to ☆

    def prepend(self, value):
        """ Insert value before LinkedList self.front.

        @param LinkedList self: this LinkedList
        @param object value: value for new LinkedList.front
        @rtype: None

        >>> lnk = LinkedList()
        >>> lnk.prepend(0)
        >>> lnk.prepend(1)
        >>> lnk.prepend(2)
        >>> str(lnk.front)
        '2 -> 1 -> 0 ->|'
        >>> lnk.size
        3
        """
        # create a new node
        # set its next to be current front
        # update current front to new node
        self.front = LinkedListNode(value, self.front)
            # right side is evaluated first and then assign to the left side
        self.size += 1
        if self.size == 0:  # or self.back is None:  (empty list)
            self.back = self.front


    def delete_front(self):
        """Delete LinkedListNode self.front from self.

        Assume self.front is not None

        @param LinkedList self: this LinkedList
        @rtype: None

        >>> lnk = LinkedList()
        >>> lnk.prepend(0)
        >>> lnk.prepend(1)
        >>> lnk.prepend(2)
        >>> lnk.delete_front()
        >>> str(lnk.front)
        '1 -> 0 ->|'
        >>> lnk.size
        2
        >>> lnk.delete_front()
        >>> lnk.delete_front()
        >>> str(lnk.front)
        'None'
        """
```

```
        assert self.size > 0, 'cannot delete from empty list'
        # make the second node the front
        # decrement size
        # maybe update back
        if self.front is self.back:
            self.back = None
        self.front = self.front.next_
        self.size -= 1

    def delete_back(self):
         # loop through nodes until current.next_ is back, or current.next_.next_ is None

    def __getitem__(self, index):   # get it by index
        """Return the value at LinkedList self's position index.

        @param LinkedList self: this LinkedList
        @param int index: position to retrieve value from
        @rtype: object

        >>> lnk = LinkedList()
        >>> lnk.prepend(1)
        >>> lnk.prepend(0)
        >>> lnk.__getitem__(1)
        1
        >>> lnk[-2]
        0
        """
        if index < -self.size or index >= self.size:
        # if not (-self.size <= index < self.size):     # triple comparison
            raise IndexError("out of bounds, invalid index”)
        elif index < 0:
            index += self.size
        assert 0 <= index < self.size
        current_node = self.front
        for _ in range (index):
            # "_" instead of i, simply use this one to increment the iteration, wont use the value
            current_node = current_node.next_
            assert current is not None, 'reached invalid index'
        return current_node.value

    def __contains__(self, value):      # operator "in"
        """Return whether LinkedList self contains value.

        @param LinkedList self: this LinkedList.
        @param object value: value to search for in self
        @rtype: bool

        >>> lnk = LinkedList()
        >>> lnk.prepend(0)
        >>> lnk.prepend(1)
        >>> lnk.prepend(2)
        >>> lnk.__contains__(1)
        True
        >>> lnk.__contains__(3)
        False
        >>> 2 in lnk
        True
        """
        current_node = self.front
        # walk along until None..
        while current_node is not None:       (while current_node)
            # check if current_node has value
            if current_node.value == value:
                return True
            current_node = current_node.next_
        # (get all the way through the linked list and no value found) got to the end, no value found
        return False

if __name__ == '__main__':
```

```
    import doctest
    doctest.testmod()
```

functional programming: method with arguments and return value, the returned value is based on some calculations, while the method does not change the object attributes.
non functional programming: method did some changes on the object attributes.

Think about extreme cases: when the linked list is empty

2 same values in linked list

For linked list, add operation at back, delete operation at back.

assert: check condition is true or not, if False, exit program and report assertion failure

```
>>> assert 0==1, 'wrong'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AssertionError: wrong
```

L.append(object)	fast
L.pop() # return and remove the last value	fast
L.insert(index, object)	slow, if we have a big list
L.pop(0)	slow, if we have a big list

LL.prepend()	fast, just look at self.front
LL.append()	fast, just look at self.back
LL.delete_front()	fast, just look at self.front
LL.delete_back()	slow, need to find 2nd last

Doubly Linked List: extra overhead, have to maintain previous reference as well, but make delete_back fast.

Week 7 • Tree

Linked List-Based vs. List-Based (linear order data structure)
—list-based Queue has a problem: adding or removing will be slow.

Check time performance: use time API start = time() end = time() - start

Non-Linear order Data Structure: Tree (hierarchical structure, recursive structure)

A tree is either **empty**, or non-empty. Every non-empty tree has a root node, connected to zero or more subtrees.
set of **nodes**(possibly with values/labels) with directed **edges** between some pairs of nodes.
one node is distinguished as **root**.
Each non-root node has exactly one **parent**.
A **path** is a sequence of nodes n_1,n_2,..n_k, where there is an edge from n_i to n_{i+1}. The **length** of a path is the number of edges in it.
There is a **unique** path from the root to each node. (in the case of the root itself, this is just n_1 if the root is node n_1)
There are no **cycles**, no paths that form loops.
Leaf: node with no children (no subtrees)
Internal Node: node with one or more children
Subtree: tree formed by any tree node together with its descendants and the edges leading to them.
Children:(of a node) all nodes directly connected underneath that node.
    # of children = # of subtrees of a node
Parent: (of a node) the one immediately above and connected to it (each node has one parent except the root, which has no parent)
Descendant: (of a node) its children, and the children of its children etc.
    The descendants of a node are its children and the descendants of its children.
Ancestor: (of a node) its parent, and the parent of its parent etc.
    The ancestors of a node are its parent, and the ancestors of its parent.

Size: number of nodes in the tree
Height: (of a tree) 1+the maximum path length in a tree.
length of the longest path from its root to one of its leaves, counting the number of nodes on the path.
A node also has a height, which is 1+the maximum path length of the tree rooted at that node.
Depth: (of a node) length of a path from root to a node is the node's depth
Arity/branching factor: maximum number of children for any node

```
""" A general tree """

class Tree:
    """ A bare-bones Tree ADT that identifies the root with the entire tree. """

    def __init__(self, value=None, children=None):
        """  Create Tree self with content value and 0 or more children

        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree] children: possibly-empty list of children
        @rtype: None
        """
        self.value = value
        self.children = children.copy() if children else []

    def __repr__(self):
        """ Return representation of Tree (self) as string that
        can be evaluated into an equivalent Tree.

        @param Tree self: this tree
        @rtype: str

        >>> t1 = Tree(5)
        >>> t1
        Tree(5, [])
        >>> t2 = Tree(7, [t1])
        >>> t2
        Tree(7, [Tree(5)])
        """
        # Our __repr__ is recursive, because it can also be called via repr...!
        return ('Tree({}, {})'.format(repr(self.value), repr(self.children))
                if self.children != []
                else 'Tree({})'.format(repr(self.value)))

    def __eq__(self, other):
        """  Return whether this Tree is equivalent to other.

        @param Tree self: this tree
        @param object|Tree other: object to compare to self
        @rtype: bool

        >>> t1 = Tree(5)
        >>> t2 = Tree(5, [])
        >>> t1 == t2
        True
        >>> t3 = Tree(5, [t1])
        >>> t2 == t3
        False
        """
        return (type(self) is type(other) and
                self.value == other.value and
                self.children == other.children)

    def __str__(self, indent=0):
        """ Produce a user-friendly string representation of Tree self,
        indenting each level as a visual clue.

        @param Tree self: this tree
        @param int indent: amount to indent each level of tree
        @rtype: str
```

```
        >>> t = Tree(17)
        >>> t1 = Tree(19, [t, Tree(23)])
        >>> t3 = Tree(29, [Tree(31), t1])
        >>> print(t3)
        29
           31
           19
              17
              23
        """
        root_str = indent * " " + str(self.value)
        return '\n'.join([root_str] +
                         [c.__str__(indent + 3) for c in self.children])

    def __contains__(self, v):
        """ Return whether Tree self contains v.

        @param Tree self: this tree
        @param object v: value to search this tree for

        >>> t = Tree(17)
        >>> t.__contains__(17)
        True
        """
        if self.children == []:
            return self.value == v
        else:
            # "in" is the recursive call
            return self.value == v or any([v in c for c in self.children])


def depth(t):
    """ Return length of longest path to the root of t

    @param Tree t: the tree to find depth of
    @rtype: int

    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> depth(tn1)
    3
    """
    # counting nodes
    if len(t.children) == 0:
        return 1  # change to 0 to count edges
    else:
        return 1 + max([depth(c) for c in t.children])


def arity(t):
    """ Return maximum branching factor of t

    @param Tree t: the tree to find arity of
    @rtype: int

    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> arity(tn1)
    4
    """
    if t.children == []:
        return 0
    else:
        branching_factors = [arity(c) for c in t.children]
        branching_factors.append(len(t.children))
        return max(branching_factors)
```

```
def leaf_count(t):
    """ Return number of leaves in t

    @param Tree t: the tree to find number of leaves in
    @rtype: int

    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> leaf_count(tn1)
    6
    """
    if t.children == []:
        return 1  # because t is a leaf
    else:
        return sum([leaf_count(c) for c in t.children])

def list_all(t):
    """ Return list of values in t.

    @param Tree t: tree to list values of
    @rtype: list[object]

    >>> t = Tree(0)
    >>> list_all(t)
    [0]
    >>> t = descendants_from_list(Tree(0), [1, 2, 3, 4, 5, 6, 7, 8], 3)
    >>> list_ = list_all(t)
    >>> list_.sort()
    >>> list_
    [0, 1, 2, 3, 4, 5, 6, 7, 8]
    """
    # implicit base case - if there are no children, then gather_lists returns []
    return [t.value] + gather_lists([list_all(c) for c in t.children])

# helpful helper function
def descendants_from_list(t, list_, branching):
    """
    Populate Tree t's descendants from list_, filling them
    in in level order, with up to branching children per node.
    Then return t.

    @param Tree t: tree to populate from list_
    @param list list_: list of values to populate from
    @param int branching: maximum branching factor
    @rtype: Tree

    >>> descendants_from_list(Tree(0), [1, 2, 3, 4], 2)
    Tree(0, [Tree(1, [Tree(3), Tree(4)]), Tree(2)])
    """
    q = Queue()
    q.add(t)
    list_ = list_.copy()
    while not q.is_empty():  # unlikely to happen
        new_t = q.remove()
        for i in range(0, branching):
            if len(list_) == 0:
                return t  # our work here is done
            else:
                new_t_child = Tree(list_.pop(0))
                new_t.children.append(new_t_child)
                q.add(new_t_child)
    return t


class Queue:
    """ A general queue """
```

```
    def __init__(self):
        """ Initialize a new empty queue """
        self._queue = []

    def add(self, item):
        """ Add item to the end of this queue """
        self._queue.append(item)

    def remove(self):
        """ Remove and return the item at the beginning of this queue """
        return self._queue.pop(0)

    def is_empty(self):
        """ Return whether or not this queue is empty """
        return len(self._queue) == 0
```

Week 8 · Traversal of a Tree

Pre-Order visit: (stack) root first, then its children (in list order)

```
    act(t)                # @param (Tree)->Any act: function to do to each node
    for c in t.children:
        preorder_visit(c, act)
```

Post-Order visit: (stack) children first, then the root.

```
    for c in t.children:
        postorder_visit(c, act)
    act(t)
```

Level-Order: (queue)

```
    to_act_on = Queue()
    to_act_on.add(t)
    while not to_act_on.is_empty():
        next_node = to_act_on.remove()
        act(next_node)
        for c in next_node.children:
            to_act_on.add(c)
```

Path: a sequence of edges (from a parent to child node) between 2 nodes
Height (of a tree): the longest path in a tree (will be between root & leaf) (from a leaf)
depth: distance (path length) from a particular node to root (from root)

Binary Tree: (arity=2) Every node has exactly 2 children, but one or both can be None.
Pre-Order: root->left child->right child
Post-Order: left child->right child->root

```
    if t is not None:
        postorder_visit(t.left, act)
        postorder_visit(t.right, act)
        act(t)
```

In-Order: left->root->right

```
    if root is not None:
        inorder_visit(root.left, act)
        act(root)  # BinaryTree.act(root)
        inorder_visit(root.right, act)
```

Level-Order:

```
    # this approach uses iterative deepening
    visited, n = visit_level(t, 0, act), 0
    while visited > 0:
        n += 1
        visited = visit_level(t, n, act)
```

```
def visit_level(t, n, act):
    """ Visit each node of BinaryTree t at level n and act on it.
     Return the number of nodes visited visited.

    @param BinaryTree|None t: binary tree to visit
    @param int n: level to visit
    @param (BinaryTree)->Any act: function to execute on nodes at level n
    @rtype: int
```

```
    >>> b1 = BinaryTree(4, BinaryTree(2), BinaryTree(6))
    >>> b2 = BinaryTree(12, BinaryTree(10), BinaryTree(14))
    >>> b = BinaryTree(8, b1, b2)
    >>> def f(node): print(node.data)
    >>> visit_level(b, 2, f)
    2
    6
    10
    14
    4
    """
    if t is None:
        return 0
    elif n == 0:
        act(t)
        return 1
    elif n > 0:
        return (visit_level(t.left, n-1, act) + visit_level(t.right, n-1, act))
    else:
        return 0
```

Binary Search Tree: binary tree where values are comparable to each other, values in left subtree are less than value of root, values in right subtree are more than value of root.

```
class BinarySearchTree:
    """Binary Search Tree class.
        represents a binary tree satisfying the Binary Search Tree property.

    === Private Attributes ===
    @type _root: object
        The item stored at the root of the tree, or None if the tree is empty.
    @type _left: BinarySearchTree | None
        The left subtree, or None if the tree is empty
    @type _right: BinarySearchTree | None
        The right subtree, or None if the tree is empty

    === Representation Invariants ===
    - empty BST: if _root is None, then so are _left and _right.
    - If _root is not None, then _left and _right are BSTs
        (can be set to empty BSTs but not None)
    - Every item in _left is <= _root, and every item in _right is >= _root
    """

    def __init__(self, root):       # constructor
        """Initialize a new BST with a given root value.

        If <root> is None, the BST is empty.

        @type self: BinarySearchTree
        @type root: object | None
        @rtype: None
        """
        if root is None:
            self._root, self._left, self._right = None, None, None

        else:
            self._root = root
            self._left, self._right = BinarySearchTree(None), BinarySearchTree(None)

        # Note that we do not allow client code to pass in left and right subtrees as parameters to the
constructor.
        # This is because binary search trees have restrictions on where values can be located in the
tree,
        # and so a separate method is used to add new values to the tree
        # that will ensure the BST property is always satisfied.

    def __contains__(self, item):           # SEARCH non-Mutating Method
```

```
        """Return True if <item> is in this BST.

        @type self: BinarySearchTree
        @type item: object
        @rtype: bool
        """
        if self._root is None:
            return False
        elif item == self._root:
            return True
        elif item < self._root:
            return item in self._left  # or, self._left.__contains__(item)
        else:
            return item in self._right

    def delete(self, item):              # DELETE Mutating Method
        """Remove *one* occurrence of item from this tree.

        Do nothing if <item> is not in the tree.

        @type self: BinarySearchTree
        @type item: object
        @rtype: None
        """
        if self._root is None:
            pass
        elif self._root == item:
            self.delete_root()
        elif item < self._root:
            self._left.delete(item)
        else:  # item > self._root
            self._right.delete(item)

    def delete_root(self):
        """Remove the root of this tree.

        Precondition: this tree is *non-empty*.

        @type self: BinarySearchTree
        @rtype: None
        """
        # if the tree consists of just the root
        if self._left.is_empty() and self._right.is_empty():
            self._root = None
            self._left, self._right = None, None
        else:
            # replace the root with the largest/rightmost value in the left subtree,
            # or the smallest/leftmost value in the right subtree
            self._root = self._left.extract_max()
```

Representation Invariant for a class:

Compared with Loop Invariant:

Week 10 · Efficiency

Redundancy:

Some programming language have better support for recursion than others; python may run out of space on its stack for recursive function calls.

The original fibonacci function was inefficient in its use of which resource — time
it was efficient in its use of another resource — space.

```
""" Fibonacci numbers """

def fib(n):
    if n <= 1:
```

```
        return 1
    return fib(n-1) + fib(n-2)

print(fib(40))

def faster_fib(n, seen):
    if n not in seen:
        if n <= 1:
            seen[n] = 1
        else:
            seen[n] = faster_fib(n-1, seen) + faster_fib(n-2, seen)
    return seen[n]

print(faster_fib(990, {}))
```

NEW fibonacci function (save computations, create a dict that you can look them up later)

__contains__ v in L (compare n times with v and items in L)
linear, time(number of steps) is proportional to n (length of the list)

sorted list: (binary search)

lg(n): base 2
the number of times I repeatedly divide n in half (by 2) before I reach 1 is the same as the number of times I double 1 before I reach/exceed n: $\log_2(n)$

lg(xy)=lg(x) + lg(y)
n = 2^(lg(n)) definition of lg(n)
2^(lg(n)) = n = 10^(log(n))

Efficiency of __contains__()
—Tree linear, proportional to n (size)
—BTNode linear
—BST lg(n) if balanced, or linear (balanced: each level except bottom is full, height is minimized)

node packing of binary tree:
n<2^h≤2n, $\lg(n) < h \leq \lg(n)+1$, where h is the minimum height of the tree to pack n nodes
If BST is tightly packed (balanced), use proportional to lg(n) time to search n nodes.

Iterative: loops in loops

Sorting (we are mainly covered 3 alg, all θ(n^2))

Bubble Sort quadratic n^2
[5, 7, 2, 3, 1, 6]
… [5, 7, 2, 3, 1, 6]
… [5, 7, 2, 3, 1, 6]

Selection Sort quadratic n^2
[5, 7, 2, 3, 1, 6]

Insertion Sort quadratic n^2
[5, 7, 2, 3, 1, 6]

Quick Sort

Merge Sort

Quick Sort

Bogo Sort

```
>>> from random import shuffle
>>> L = [1, 2, 3, 4, 5, 6]
>>> shuffle(L)
>>> L
[6, 5, 4, 3, 1, 2]
```

Quick Sort: partition and concat nlg(n)

```
qs([4, 2, 6, 1, 3, 5, 7])
qs([2, 1, 3]) + [4] + qs([6, 5, 7])
qs([1])+[2]+qs([3]) + [4] + qs([5])+[6]+qs([7])
[1] + [2] + [3] + [4] + [5] + [6] + [7]
[1, 2, 3] + [4] + [5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
```

radix sort O(n) time proportional, sort bounded integers

Definition of big-Oh:

$$O(1) \subseteq O(\lg(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(n^n)$$

Divide & Conquer:
divide into smaller problems; solve subproblems recursively; combine to get result.

mergesort → more combining work
quicksort → more dividing work
https://en.wikipedia.org/wiki/Sorting_algorithm
quick sort: average-case O(nlogn), worst-case O(n^2)
merge sort: average-case O(nlogn), worst-case O(nlogn)
bubble sort: average-case O(n^2), worst-case O(n^2)
insertion sort: average-case O(n^2), worst-case O(n^2)

Week 11 ·

Sequence, Loop, Condition & efficiency
Sequence (max)
Loop (count and multiply)
Condition (consider each branch separately)

lg(n): this is the number of times you can divide n in half before reaching 1.
a**b=c means $\log_a c=b$
This runtime behavior often occurs when we divide and conquer. (binary search)

Assume lgn(log base 2) but the difference is only a constant.
$2^{\log_2 n} = n = 10^{\log_{10} n}$ $\log_2 n = \log_2 10 \times \log_{10} n$ SO we just say O(lg n)

Hash immutable objects, cannot has mutable objects.

store information about object o at index hash(o)%n, $0 \le hash(o) < n$

Hash Function: any function that can be used to map data of arbitrary size to data of fixed size
Perfect Hash Function for a set S is a hash function that maps distinct elements in S to a set of integers, with no collisions. In mathematical terms, it is a total injective function.

Collisions: even a well-distributed hash function will have a surprising number of collisions.

Closed Addressing/Chaining: store a linked list at each bucket and insert new ones at the head
Open Addressing/Probing: each array element to contain only one key, but to allow keys to be mapped to alternate indices when their original spot is already occupied
Hash Table:
search average-case O(1), worst-case O(n)
insert average-case O(1), worst-case O(n)
delete average-case O(1), worst-case O(n)

Collision: if $|U|>m$, for all L, there exists distinct $k_1, k_2 \in U$, $h(k_1)=h(k_2)$
Perfect Hash Function: if $m \ge |U|$, there are at least as many array slots as possible keys, then there always exists a hash function h which has no collisions.

load factor = number of entries / number of buckets (slots)

A constant time operation is one whose running time does not depend on the size of the input.
All Python arithmetic and comparison operations, variable/attribute access and assignments, and print and return all take constant time.